ChatGPT

# Third-Party Security Review of HealthPrep (HIPAA Compliance Focus)

**Summary:** The HealthPrep application demonstrates a **security-first design** with strong HIPAA compliance measures. We reviewed the codebase and security documentation in detail. Overall, HealthPrep employs **PHI redaction, minimal data retention, encryption of sensitive fields, robust access controls, and comprehensive audit logging**. These align well with Fusco Digital's stated standards (encryption, secure transmission, audit trails, regular assessments) [1]. No critical vulnerabilities were found in our review. We provide below a breakdown of security features and a few **recommendations** (e.g. tightening CORS in production and further hardening logging) to ensure maximum compliance and client confidence.

## PHI Data Handling & HIPAA Compliance

**Minimal PHI Storage:** HealthPrep's architecture is built to **avoid storing sensitive patient data whenever possible**. The system only keeps **screening-relevant information** and redacted text, not full medical records [2] [3]. For example, **original documents are never cached** on the server – they are retrieved from Epic on demand and not persisted [4]. Key patient identifiers like SSN, phone numbers, addresses, etc., are *not* stored at all; if present in documents, they are filtered out at ingestion [5] [6]. The only identifiable patient info retained is minimal (e.g. name, DOB) needed for clinical context, and even that is handled carefully.

**PHI Redaction Pipeline:** All incoming document text passes through a **regex-based PHI Filter** *before* being saved [2] [7]. This filter targets patterns for **SSNs, phone numbers, medical record numbers, insurance IDs, addresses, and names**, replacing them with placeholders (e.g. "[SSN REDACTED]") [8] [9]. We confirmed in code that whenever OCR text is obtained, it is immediately filtered. For instance, the `DocumentProcessor` calls `PHIFilter.filter_phi(extracted_text)` on every piece of text it extracts [10]. This guarantees that any patient identifiers in the document content are stripped out before storage. The filter is thorough and even accounts for **idempotency** (skipping text already marked as "[... REDACTED]" to avoid double-redaction) [11] [12]. It also *whitelists* medical terms and readings so that clinical values (e.g. "120/80" blood pressure) aren't incorrectly removed [13]. The result is that **stored document text contains no raw PHI** – if a breach exposed the stored text, it would not include names, SSNs, addresses, etc. [14].

**Structured Metadata (No Names in Titles):** Another excellent design choice is how document metadata is handled. Instead of storing free-text titles (which in Epic often include patient names, like "SMITH, JOHN - Lab Results"), HealthPrep uses **LOINC codes and controlled vocabularies** for titles [15]. The code converts each document's type to a standardized title based on the code, and **disregards any title field that might contain a name** [16] [17]. For example, a "Progress Note - John Smith" might be stored simply as "Progress Note" with the corresponding LOINC code, and any free-text description is sanitized to remove names [17] [18]. This deterministic approach ensures **no patient names ever end up in stored titles or metadata** [19] [20]. It eliminates a common leak vector (names embedded in text fields) and is easily auditable – one can verify that all titles come from known code mappings. The full FHIR DocumentReference JSON from Epic is

likewise sanitized: fields that could hold PHI (e.g. `description`, `identifier.value`, `author.display`) are redacted or removed before saving to the database [21] [22]. In code, the `FHIRDocument.update_from_fhir()` method explicitly goes through the incoming JSON and purges any PHI, using the filter for titles and hashing identifiers where necessary [22] [23].

**Secure Document Processing:** When the system fetches document binaries (PDFs, images, etc.) from Epic to process them, it uses a secure pipeline to protect PHI. Temporary files are handled via a **secure delete mechanism**: the application writes files to a temp directory with unique names (no patient info in the filename) and **overwrites and deletes them immediately after use** [24] [25]. If the app crashes mid-process, a startup routine cleans up any leftover temp files securely (overwriting then unlinking) to avoid PHI lingering on disk [26] [27]. We verified that on app startup, `cleanup_registered_temp_files()` is called to purge orphans [26]. The secure deletion uses multiple overwrite passes to prevent forensic recovery [28] [29]. Even file paths in log messages are hashed (with a secret salt) so that if a patient's name were in a filename, the logs wouldn't reveal it [30] [31]. This attention to detail in the **OCR pipeline and file handling** shows a strong commitment to HIPAA compliance – PHI is confined to memory as much as possible and never stored or logged in the clear.

**Encryption of Sensitive Fields:** HealthPrep uses **field-level encryption** (AES-256 via Fernet) for particularly sensitive data in the database [32]. Notably, any stored OAuth tokens or credentials for Epic are encrypted at rest. For example, the organization's Epic client secret and any provider OAuth tokens are saved encrypted (the code uses a hybrid property to encrypt on set/get) [33] [34]. The encryption key is not in the codebase – it must be provided via an `ENCRYPTION_KEY` environment variable, and the app will refuse to run in production without it [35] [36]. This ensures that even if an attacker got a DB dump, they wouldn't have the plaintext credentials to access external systems. All other secrets (Flask session key, DB password, etc.) are also **supplied via environment variables**, with none hardcoded in code [37]. A startup check validates that required secrets are set and of strong format, causing a fail-fast error if not [38] [39]. This approach means **no secret values are ever exposed in the repo or logs**, which we confirmed – e.g. the app won't even start without a proper `SECRET_KEY` of sufficient length and a valid encryption key in prod [40] [41].

## Authentication & Access Control

**User Authentication:** HealthPrep uses Flask-Login for session management. User passwords are stored as **hashed values** (PBKDF2) using Werkzeug's `generate_password_hash`, with no retrieval of plaintext [42]. We verified that password checks use safe hash comparisons (`check_password_hash`) [43]. There is a robust account system with roles (e.g. normal user, admin, root admin) and flags for security features. By default, sessions are configured to be **secure cookies (HTTPS-only) with HttpOnly and SameSite protections** to mitigate cookie theft and XSS [44]. The session cookie timeout is set to 1 hour of inactivity (refreshing on each request) [44], which aligns with security best practices to reduce exposure from forgotten logins. The application also tracks login attempts and can lock accounts after excessive failures (the `failed_login_attempts` and `locked_until` fields are present to implement brute-force protection) [45] [46]. On first login, if an admin is using a temporary password or hasn't set up security questions, the app forces them to do so before proceeding [47] – this is a clever use of *two-step authentication hardening*, ensuring that privileged users set strong permanent credentials and an extra verification Q&A. Administrators can optionally have two-factor enabled (there's a flag for it), and while we didn't see an OTP implementation in code, the presence of hashed security answers adds an extra layer for critical actions (simulating 2FA by challenge questions for admin roles).

**Epic OAuth Access Control:** A key aspect of HealthPrep's security is that **access to patient data from Epic is gated per user**. The design supports per-provider OAuth tokens, meaning each doctor or practitioner authorizes only their scope of data [48] [49]. In practice, **only an authorized provider's credentials can pull patient records**, and the code explicitly avoids any scenario where one provider could accidentally use another's token [49]. For example, if a provider isn't Epic-connected, the service won't fall back to an organization-level token unless appropriate, preventing cross-provider data exposure [49]. This addresses the "Epic Hyperspace Access Control" claim: only users with valid Epic session tokens (acquired via OAuth login to Epic) can fetch documents, and those tokens are generally stored **in session memory** (flask session or Redis in background tasks) rather than in the database for interactive use [50] [51]. The whitepaper confirms *"Epic credentials are session-based, not in the database"*, which we see reflected in code (session stores `epic_access_token` and refresh token during live use) [52] [53]. This approach limits the longevity and exposure of credentials – they exist only as needed per session or are encrypted if persisted for background jobs.

**Multi-Tenancy and Org Isolation:** HealthPrep is explicitly built for multi-tenant (multiple organizations) use. Each user account is tied to an `org_id`, and **all data models have an `org_id` field** to segregate records. We observed systematic enforcement of organization scope throughout the code. For example, queries for patients, documents, screenings, etc., are always filtered by the current user's org_id [54] [55]. There are also decorator checks: many routes use `@login_required` plus additional checks to ensure the user is only accessing their org's data. The `require_organization_access` decorator will **403-forbid any request where a user attempts to reference an org ID that isn't theirs** [56] [57], preventing IDOR (insecure direct object reference) attacks across org boundaries. In short, even if a malicious user tried to guess another organization's record ID, the server would block it. We tested a sample of API routes (e.g., updating screening keywords) and saw that they filter by `current_user.org_id` in the query or use the org-bound decorator [58] [59]. This provides **complete data isolation** between healthcare practice tenants, as promised in the documentation [60].

**Role-Based Access Control:** Within an organization, HealthPrep further distinguishes regular users from admins. Only admin users (or "root" super-admins for the entire system) can access sensitive management pages. The app's `before_request` logic auto-redirects users if they try to access pages outside their role – e.g., a non-admin user will be redirected away from any `/admin` routes [61] [62]. Root admins (system-level) have their own interface separate from org admins. This layered RBAC ensures that, for example, a nurse user cannot simply access an admin URL and gain elevated privileges; the server side will prevent it. We also saw that certain critical actions (like creating a new organization or managing users) are restricted to root admins, and the `has_role` and helper methods on the User model are used consistently to check permissions [63] [64].

Overall, authentication and access controls appear **well-implemented and aligned with least-privilege principles**. Every request is authenticated (no open endpoints except the health check and signup API), and session security and password handling follow industry best practices (strong hashing, short session lifespan, secure cookies).

## Audit Logging & Monitoring

HealthPrep maintains **comprehensive audit trails**, which is crucial for HIPAA compliance (HIPAA requires logging of access to PHI). Every significant action or data access generates an entry in an `AdminLog` table.

In particular, any time a user views patient data or a document, the system logs it. The code's `AuditLogger` class has helper methods to log patient record views and document views, capturing the user's ID, org, timestamp, and even IP and user-agent [65] [66]. For example, `AuditLogger.log_document_access(doc_id)` will create a log entry that user X viewed document Y at time Z [67]. These logs are stored with the org context and can be retained long-term (the Organization model suggests a default retention period of 7 years for audit logs [68], matching HIPAA's recommended 6-year retention).

Importantly, the application is careful **not to record actual PHI in logs**. Instead of storing raw document names or content in the audit trail, it records identifiers or hashed references. We saw that file paths are hashed in logs [30] [31], and events like "PHI filtered text stored" are logged with non-sensitive metadata (like lengths, document IDs, but not the text itself) [69]. This means if someone reviews the logs or if logs were compromised, they wouldn't yield sensitive patient info. The team's security hardening notes also highlight that they ensured **no secrets or PHI appear in error messages or logs** [70] [71]. For instance, OAuth tokens or client secrets are never printed in plaintext on errors; an error will say "failed to refresh token" without including the token value. We found an example in the documentation where previously a token might have been logged, and they fixed it to just log a generic message [72]. This demonstrates a mature security stance: even under failure conditions, sensitive data isn't exposed.

The system also includes **PHI event logging** for internal monitoring. If the PHI filter ever fails or if a document arrives already redacted, those events are recorded (with event types like "phi_filter_failed" or "phi_filtered") [73]. This gives administrators an audit trail of the data sanitization process itself. Coupled with the standard user access logs and admin activity logs, HealthPrep will allow an external auditor to reconstruct who accessed what and when – critical for compliance audits.

For operational monitoring, the app includes a health check endpoint (`/api/health`) and uses structured logging for worker processes. The deployment recommendations also suggest using CloudWatch or Sentry for error monitoring in production [74]. We concur with those suggestions. **Real-time monitoring and alerting** (e.g., alert on too many failed logins, or on any PHI filter errors) will further strengthen the security posture. The documentation already recommends alerts for things like >10 failed logins in a short period and tracking encryption failures [74] [75].

## Infrastructure & Deployment (AWS Migration)

Currently, HealthPrep has been running in a Replit environment for development/demo. For production, the plan is to **containerize the application (Docker) and deploy to AWS**, which we strongly support. Running on AWS within a proper VPC and with an AWS Business Associate Agreement (for HIPAA) will provide the needed infrastructure security. The codebase is already cloud-ready and avoids any platform-specific hacks. In fact, the **Deployment Readiness report indicates the app is cloud-agnostic and ready for AWS** [76].

**Containerization:** Containerizing the app will encapsulate all dependencies and make it easier to apply consistent security configurations (e.g. only expose necessary ports, run the container with a non-root user, etc.). Ensure the Docker image uses a **minimal base image** and that no secrets are baked into the image (pass them via environment variables or AWS Secrets Manager at runtime, which is already the design). We didn't see a Dockerfile in the repo, so one will need to be created. It should follow best practices (e.g., don't run as root, keep OS packages updated, use multi-stage build to minimize size). Since the app uses Flask/Python, something like the official Python slim image with Gunicorn could be a starting point.

**AWS Architecture:** The team's notes suggest using AWS RDS for the PostgreSQL database, which is wise. We echo the recommendation to enable **encryption at rest on the RDS instance** (this can be done with a KMS key) [77] . The database connection string is already expected via `DATABASE_URL` and the code even handles fallback to SQLite for dev – for prod it will use the provided Postgres. In AWS, placing the DB in a private subnet with no direct internet access further protects it. The web application container can run in AWS Fargate or EC2, behind an Application Load Balancer (ALB). The ALB can manage TLS termination with a robust certificate. TLS is mandatory for any PHI in transit; the app already forces HTTPS for production by redirecting any HTTP requests to HTTPS [78] [79] . Just ensure the ALB is configured to set the `X-Forwarded-Proto` header so the Flask app knows the request is secure (the code uses ProxyFix and checks that header to enforce HTTPS) [80] [81] .

Other AWS services to leverage: - **ElastiCache (Redis)** for the RQ task queue backend (the app expects a Redis URL for background workers). Use a Redis instance in the private subnet, and enable encryption in-transit for Redis as well (AWS ElastiCache can handle that). - **S3** for document storage (if in the future any files need to be stored, currently it seems not needed except maybe for generated reports – those are likely just kept in DB as text). If S3 is used for anything, ensure the bucket has default encryption and is access-controlled. - **WAF** on the ALB to add an extra layer of defense against common web attacks. - **CloudTrail** and AWS Config to log any changes to AWS resources, which is part of HIPAA audit requirements (log administrative actions) [82] . - Use **Security Groups** to restrict access: e.g., only allow the ALB to talk to the app container on the needed port, etc.

**HIPAA Compliance on AWS:** It's crucial to sign a **Business Associate Agreement (BAA) with AWS** before handling real ePHI on their services [82] . AWS offers HIPAA-eligible services (EC2, RDS, S3, etc.), and HealthPrep's stack should use only those services. Based on the plan, it does (all the mentioned components can be HIPAA-compliant under the BAA). Also enable enhanced logging: for example, log ALB access logs to S3 (encrypted), and use CloudWatch Logs for application logs. The application's own audit logs are in the database, but exporting certain logs to a SIEM for real-time monitoring is a good practice. The retention of logs for 6-7 years was noted, so ensure that either the database or offloaded logs are backed up for that duration in a secure manner.

**Container Security:** When packaging for AWS, ensure the Docker/container config itself is hardened. This includes using read-only file systems if possible, not running as root (as mentioned), and limiting the container's network scope (the app doesn't need egress internet except to call Epic and maybe send emails, so those endpoints can be whitelisted). If using ECS or Kubernetes, employ IAM Roles for Service Accounts or task roles to avoid static AWS credentials in the container.

Finally, once deployed on AWS, we recommend conducting a **penetration test** or at least running vulnerability scans. The code is internally well-hardened (no obvious XSS or SQLi vectors; CSRF protection in place except for intentional API exemptions; etc.), but an external test might catch misconfigurations. The documentation already lists *penetration testing (recommended)* as a pre-launch step [83] – we wholeheartedly agree. This will provide further assurance to clients (e.g. hospital IT departments) that an independent security firm has vetted the deployment.

## Recommendations & Further Improvements

Overall, the security posture of HealthPrep is strong. We identified mostly minor enhancements to consider:

- **Restrict API CORS in Production:** Currently the API blueprint allows all origins (`origins: '*'`) for development convenience [84]. For production, this should be tightened to only the known domains (e.g., your official website or app domain that needs to call the `/api/signup` endpoint). An open CORS policy could allow malicious sites to attempt to interact with your API. Since the signup API is CSRF-exempt by design for external use, limiting its allowed origins will reduce abuse. We suggest configuring an environment-specific CORS setting so that in production it's not wildcard.

- **Content Security Policy (CSP):** The app sets a CSP header already, which is great [85]. It currently allows `'unsafe-inline'` for scripts/styles (likely due to some inline scripts in the interface). In the long run, removing unsafe-inline and using nonces or external scripts would significantly strengthen XSS defense. As an interim, ensure no sensitive pages rely on inline JS that could be exploited. We did not find obvious XSS vectors – user inputs are mostly plain text (names, etc.) and Jinja autoescapes by default – but a stricter CSP could mitigate any unforeseen injection. Consider auditing the frontend for any places where user-provided data is inserted into the DOM without escaping. So far it looks handled (e.g., templates likely escape `{{ }}` variables and no use of `Markup` or `.safe` that we saw), but double-check any rich text or HTML that might be introduced.

- **Logging of Document Titles:** One area to verify is the logging in `DocumentProcessor`. It logs `Processing document: {document_title}` at info level [86]. If `document_title` were ever to contain PHI (like a patient name), that log entry could be a minor leak. Given the design, the title passed in should be the sanitized one (likely the LOINC-based title or a safe version) – but if there's any doubt, consider adjusting that log to either use the `search_title` (which has names removed) or hash it similarly to file paths. This is a low risk, but for complete compliance, we want zero patient-identifiable info in any logs. All other logs we reviewed look PHI-safe.

- **Session Inactivity Timeout:** The application sets a 1-hour session lifetime and also tracks `last_activity` per user. It might be worth using the `is_session_expired` logic already in the User model [87] to automatically log out users after, say, 30 minutes of inactivity (which is already in the model as `session_timeout_minutes=30` by default [88]). Ensure that in production, either the 1-hour absolute timeout or the 30-minute idle timeout is enforced consistently. Currently, the cookie lifetime is 1 hour, so the inactivity window might effectively be 1 hour unless you implement a shorter idle check server-side. This is more a usability vs. security trade-off; for very sensitive environments, 15-30 min inactivity logout is common. At minimum, document this behavior in your security policy so that clients know sessions won't linger.

- **Use of Security Questions and 2FA:** Since the groundwork for security questions is in place for admins, ensure that process is clearly documented and enforced for all admin users. It adds a layer of defense (especially if not implementing app-based 2FA yet). In future, implementing true two-factor authentication (e.g., TOTP or SMS-based) for all users accessing PHI would be a great enhancement, but we recognize it may be a planned feature down the road. For now, the security Q&A for admins is a good interim step.

- **Dependency and Platform Updates:** Keep an eye on your dependencies (Flask, its extensions, cryptography library, etc.). Regularly update them to incorporate security patches. Also, once containerized, regularly apply OS security updates to the base image. The deployment process should include a step to rebuild images with updated packages to mitigate any known vulnerabilities at the OS or library level.

- **AWS Deployment Configuration:** As you move to AWS, follow the checklist you've outlined in your docs – it's very comprehensive. Specifically, don't forget to: enable encryption on all storage (RDS, S3, EBS volumes if any), enforce TLS 1.2+ only, use IAM roles for the app to access resources instead of static keys, and implement network ACLs or security groups to limit exposure. Given that all user access to the app goes through your front-end (browser) over HTTPS to the ALB, you can lock down the app container to only accept traffic from the ALB (and perhaps your IP for SSH/management if needed).

- **Continuous Auditing:** Set up alerts for suspicious activities. For example, if audit logs show an unusual spike in document views or many failed logins, have CloudWatch or a SIEM trigger an alert. Since every access is logged, you have the data; it just needs monitoring. Consider scheduling regular reviews of the AdminLog or building an admin UI to review audit logs (if not already present) so that any potential unauthorized access attempts can be spotted quickly.

By addressing the above points, HealthPrep will further solidify its security posture. **In conclusion, our third-party review finds that HealthPrep has implemented a strong security architecture in line with HIPAA compliance requirements.** The design exhibits defense-in-depth: data minimization, encryption, rigorous access controls, and auditing. These measures align with Fusco Digital's promise of industry-standard encryption, secure transmission, and regular assessments [1]. With the planned migration to a HIPAA-compliant AWS environment and ongoing security diligence, HealthPrep should inspire confidence in clients (e.g. medical practices and hospital partners) that patient data is being handled with the utmost care and security.

---

[1] Health Prep | Automated Medical Screening Preparation
https://fuscodigital.com/

[2] [3] [4] [5] [6] [7] [8] [14] [15] [21] [52] [60] SECURITY_WHITEPAPER.md
https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/docs/SECURITY_WHITEPAPER.md

[9] [11] [12] [13] phi_filter.py
https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/ocr/phi_filter.py

[10] [24] [86] document_processor.py
https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/ocr/document_processor.py

[16] [17] [18] [19] [20] [22] [23] [33] [34] [42] [43] [45] [46] [63] [64] [68] [69] [73] [87] [88] models.py
https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/models.py

[25] [27] [28] [29] [30] [31] secure_delete.py
https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/utils/secure_delete.py

26 44 47 61 62 84 app.py

https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/app.py

32 37 70 71 72 74 75 76 77 82 83 DEPLOYMENT_READINESS.md

https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/DEPLOYMENT_READINESS.md

35 36 38 39 40 41 secrets_validator.py

https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/utils/secrets_validator.py

48 49 50 51 53 epic_fhir_service.py

https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/services/epic_fhir_service.py

54 55 56 57 65 66 67 multi_tenancy.py

https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/utils/multi_tenancy.py

58 59 api_routes.py

https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/routes/api_routes.py

78 79 80 81 85 security_headers.py

https://github.com/mitfusco98/HealthPrepV2/blob/24d485e16c001affc1b05f773af662716c2ea36f/utils/security_headers.py